The Design and Use of Synchronous Collaborative Software Engineering Tools

Carl Cook Warwick Irwin Neville Churcher

Technial Report TR-05/05, July 2005
Software Engineering & Visualisation Group,
Department of Computer Science and Software Engineering,
University of Canterbury, Private Bag 4800,
Christchurch, New Zealand
{carl, wal, neville}@cosc.canterbury.ac.nz

The contents of this work reflect the views of the authors who are responsible for the facts and accuracy of the data presented. Responsibility for the application of the material to specific cases, however, lies with any user of the report and no responsibility in such cases will be attributed to the author or to the University of Canterbury.

This technical report contains a research paper, development report, or tutorial article which has been submitted for publication in a journal or for consideration by the commissioning organisation. We ask you to respect the current and future owner of the copyright by keeping copying of this article to the essential minimum. Any requests for further copies should be sent to the author.

Abstract

We have developed a framework, CAISE, to support Collaborative Software Engineering (CSE). In this paper, we focus on the development of CSE tools within the CAISE framework. We present examples to illustrate how such tools are constructed and how they support real-time multi-user collaborative software development. We also address issues related to the number of collaborators and discuss performance aspects.

Contents

1 Introduction

Software engineering is predominantly a collaborative activity, where typically multiple teams of people develop several versions of a range of products at any one time. Surprisingly, tools to support synchronous, or real-time Collaborative Software Engineering (CSE) are still restricted to minor tasks for specific software engineering purposes—if they make it out of the research prototype stage at all.

Today there is a real need for CSE tools, and this demand will grow as software engineering becomes an increasingly complex and heterogeneous discipline. While support for collaboration has emerged in other areas of everyday applications such as file sharing, instant messaging, and generic tele-working, software engineers themselves appear ambivalent about the opportunities and potential benefits of more comprehensive tool support. Accordingly, research into CSE is both timely and imperative.

The main premise of our research is that by enabling fine-grained CSE through seamlessly integrated tool support, it is possible to raise the very restricted levels of communication within current software engineering practice. The value of active communication has long been recognised in Computer Supported Collaborative Work (CSCW) research and user interface evaluations, and is re-emerging in software engineering within the eXtreme Programming movement. Increased programmer communication, as put forward current CSE research, is likely to produce more informed decisions during the development stage of software engineering, and less likelihood of costly coding conflicts.

In this paper, we present tools developed as part of an ongoing investigation into the possibilities for real-time CSE. We describe CAISE, a framework to support the development of CSE tools, and discuss how new tools can be constructed within the CAISE framework to support the real-time development of a collaborative software project. We also address issues related to large group sizes and performance aspects of the framework.

The remainder of this paper is structured as follows. In section 2 we provide a background on recent work related to CSE. This section also discusses reasons as to why new tools have not been as prolific within CSE research when compared to other areas of Computer Science, and why commercial IDE vendors appear slow to adapt new collaborative features into mainstream software engineering tools. Section 3 presents our framework which supports the development and runtime requirements of new CSE tools. This section also presents two typical CSE tools constructed from our framework.

In section 4 we provide details on how to implement new CSE tools within the CAISE framework. This section includes discussions on concurrency control, tool synchronisation, and the API that tools use to access the CAISE framework's services. Section 5 discusses how CAISE tools can be used with varying degrees of group development activity. Section 6 then gives details on performance analysis of the framework. Conclusions and further work are presented in section 7.

2 Background

While the proposal of tools to support CSE often draws an enthusiastic response from practitioners, the design and implementation of commercial-strength tools is a challenging task. Even once such tools have been developed, there is no guarantee that they will gain widespread adoption; this mistake has been made within related areas of Computer Supported Collaborative Work (CSCW) research [3].

Any CSE tool has complex issues to address, such as user interface design, CSCW control and management, varying levels of collaboration requirements and expectations between developers within a group, and support of multiple artifact types with possibly multiple views of each artifact type. There are also technical aspects to address such as concurrency control and distributed systems, along with the standard software engineering technicalities such as parsing, semantic modelling and source code management.

There have been some very sound prototype tools and facilities for CSE produced from within the field of research, but very few of these concepts have evolved into components within professional tools. A significant difficulty is that conventional software engineering tools are based on a single-user design, and 'bolting on' collaborative features doesn't necessarily scale or provide the level of improvement envisaged.

For example, a single-user tool can be extended to support distributed collaborative code editing or collaborative UML diagramming. This 'Groupware' approach of augmenting single user tools with CSCW functionality is suitable for generic trivial applications, but for software engineering tools the domain is significantly more demanding. Consider the case where both modes of work were to be supported collaboratively at the same time. Implementing round-trip engineering in conventional tools is complex, but to implement collaborative round trip—or multiple view—software engineering tools is a substantially more difficult task. To support such complex functionality as collaborative distributed round-trip software engineering, often the only means is by completely redesigning tools upon a foundation of collaborative work technology.

While it is certainly possible to implement collaboration-enhanced software engineering tools, the single significant barrier to the success of tools may possibly be the poor ratio of tool power versus development effort.

2.1 Related Work

In the last year many of the major commercial IDEs have taken significant steps towards code-level real time collaboration. Of the five Java IDEs that have the largest market shares, Eclipse, Borland and JSE now support shared development facilities, and all environments are promising more to come in the next major releases.

Aside from fully featured IDEs, many specialist tools support collaborative modes of work. Poseiden enterprise edition, for example, allows the authoring in real time of UML documents by any number of users in a distributed setting [2]. There are also collaborative plug-ins available for Oracle and Rational's IDEs, bringing them into the market for code-level collaborative development tools.

Within the field of research, there are also numerous specific and ambitious software engineering tools to accommodate a range of tasks. For collaborative

change impact reporting the Palantir architecture exists [13]. To visualise the activity of large shared code bases, the Augur visualisation suite may be used [10]. For web-based shared UML editing, Rosetta is a well known tool [11], and for distributed eXtreme Programming a new development tool called Moomba has been released [12].

For a detailed annotated bibliography on the CSE tools mentioned above, and related areas of research, please refer to [4].

2.1.1 Comparison to CAISE

CAISE differs from the majority of other CSE research projects in three ways. Firstly, it has a holistic approach in that the entire infrastructure is based upon collaboration. The CAISE server, for example, exists simply as a shared IDE engine for collaborative tools, as depicted in figure 1. CAISE is not simply a collaborative add-on project to existing single-user tools.

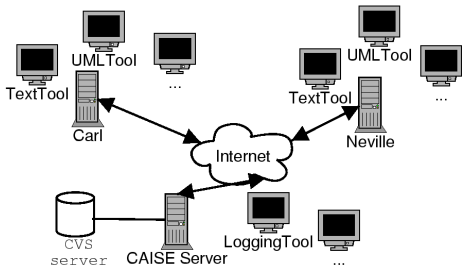


Figure 1: A general schematic representation of the CAISE architecture.

The CAISE framework also differs by supporting rather complex CSE tools. It is not restricted to particular tasks or methodologies—there is no theoretical limit to the scale and ability of CAISE-based tools.

Finally, CAISE is implemented as a framework rather than a tool-set, as described in [9]. We are well aware that different programmers will have different tool requirements. Accordingly, we make no assumptions about the 'right' set of software engineering tools. Instead, we have focused on designing a framework that can support a wide range of custom collaborative applications.

3 The CAISE Architecture and Example Tools

3.1 Architectural Overview

The authors of the Concurrent Versioning System (CVS) say *"CVS is no substitute for communication"* [1]. We concur, and know of no other code repository systems that support communication any better. Therefore, the basis for the CAISE set of collaborative software engineering tools was to allow programmers to work collaboratively without sacrificing communication. CAISE-based tools achieve this by keeping all programmers within a group synchronised in real time, at the same time providing customisable user awareness and project state information to the individual tools.

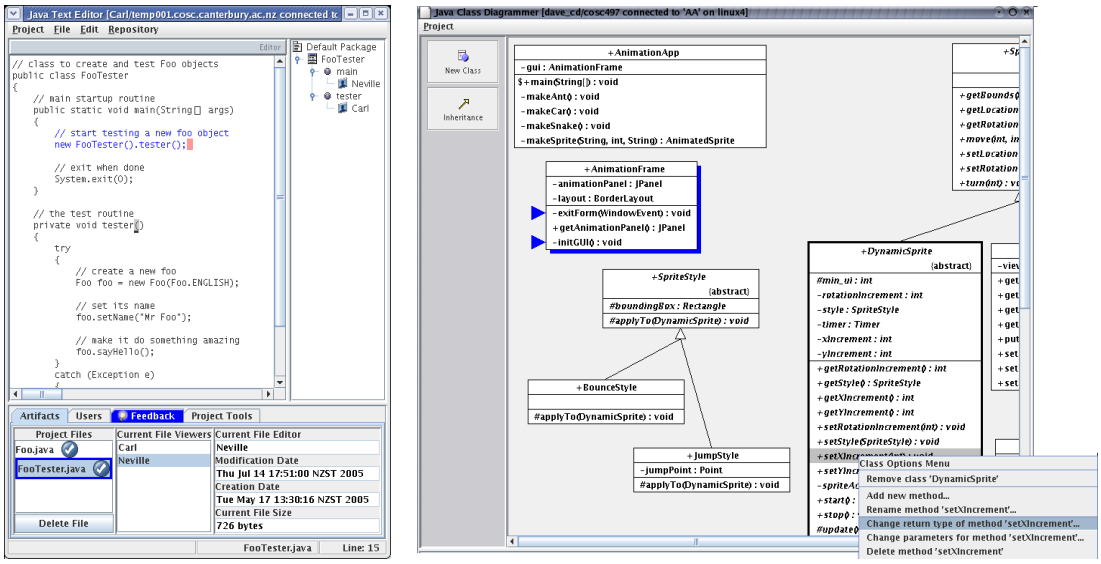
The CAISE architecture and supporting tools do not aim to replace systems such as CVS; the ability to work in private at times and to be able to keep

different versions of programs separate are elements that very few programmers could do without. Our tools are designed support what code repositories do not provide: communication between developers and tools during fine-grained real time collaboration, such as multi-user coding within the same file. Additionally, the CAISE infrastructure does not impose a specific methodology onto CAISE-based tools; tool developers can implement particular methodologies on top of the CAISE architecture if and when required.

The CSE tools presented in this paper are of a realistic scale and support the core functionality expected of any software engineering suite, including project compilation and execution support, editor undo, cut, copy and paste, UML class diagramming, and round-trip engineering between tools. As the CSE tools are capable of supporting realistic software engineering tasks, we are now in a position to investigate CSE in detail.

To date, the current set of CSE tools have performed well anecdotally [6], empirically [7], and heuristically [8]. The underlying CAISE architecture, as recently described in [9], allows for the rapid development of fully featured CSE tools, such as those presented in this paper. Typically, CAISE tools are designed to support patterns of collaboration evident in software engineering practice. Such patterns are presented in [5].

3.2 Tool Features



(a) A Java code editor. When viewed in colour the remote text highlighting and tele-carets are visible.

(b) A UML class diagrammer. Remote user positions are indicated by blue markers.

Figure 2: CAISE-based development tools with CSCW awareness support.

Since their conception, we have been constantly refining the CSE tools discussed in this paper in order to provide realistic Software Engineering environments. These tools are presented in figure 2. The main features currently supported include:

- Round-trip engineering between all tools.
- Full multi-user editing and UML class diagramming capabilities with a *relaxed WYSIWIS* view, including collaborative undo.
- An artifacts panel that displays the current compilation state of each artifact as well as editor details and file information.
- A code editor which provides remote modification highlighting and *telecarets*. The diagrammer indicates remote developer locations through special markers and tool-tips.
- Instant messaging and an audio chat channel.
- All relevant aspects of the user interface have been designed to accommodate the constantly changing state of each developer's display.
- A source code control system has been integrated to allow a CAISE project to access a central code repository.
- Build and run facilities, including protection from crosstalk when attempting to compile during times of high development activity
- Event-based collaborative feedback information, such as Degree of Interest (DOI) reports relating to other user locations within the project, and model change events as the project evolves.
- A collaborative *User Tree* that provides a model-based view of developers within the project.

There are numerous other features that have been built into other tools such as code-age highlighting, as well as stand alone graphical components such as a real time project change graph. These features have been presented in previous papers [6, 9], which focused primarily in describing the CAISE infrastructure.

The majority of the features built in to the above tools, such as the artifacts panel and user tree, are stand-alone components made available from the CAISE client widgets library. These components can be utilised from within any application, and applications that use such components do not require any specific software engineering knowledge or capabilities. For example, a stand-alone text editor can be enhanced by incorporating the CAISE collaborative user tree into its user interface. Given that the text editor conforms to the CAISE tool protocol as specified in section 4.3, the user tree will highlight the method, class and package that the editor is currently modifying, without the editor needing to possess any specific software engineering capabilities.

3.3 Life-cycle of an Artifact

Before presenting the design and implementation of CAISE and its associated software engineering tools, it is worthwhile discussing the typical life-cycles of artifacts within a software project.

3.3.1 Project Branches and File Versions

For any realistically-sized software project, including CAISE-based projects, it is likely that multiple versions are stored within a source code repository system. This is regardless of the number of programmers or development methodology followed. By branching, as illustrated in figure 3, trivial modifications to files within a previous release of an application can be made immediately upon request, regardless of the compilation state of the current version of the project. Such changes can then be integrated into the main version trunk once the main version is in a compilable and stable state, instead of attempting to rush the construction of the current project that incorporates the requested modifications.

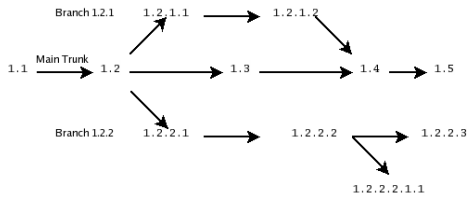


Figure 3: A typical revision history tree for a software project.

Branching, where a complete set of project files is duplicated for an alternate stream of development, does not necessarily have to be undertaken within a software project. Within a single development trunk, however, source code repository systems will still checkpoint sequential versions of all project files; typically this is done automatically upon each commit of updated files back into the central repository. Accordingly, it is important to realise that source code repository systems have the ability to produce a previous version of any file, possibly replacing the current version if required.

Given that multiple branches of a project may exist, and that each file within a branch has a potentially large number of previous versions, we now turn our focus to the life-cycle of a file in the context of a single version.

3.3.2 File Merging

Any given version of a single file may be modified by several developers concurrently. Within the CAISE framework, the modifications are made on the one instance of the file, which in many ways simplifies the file-sharing process. When using conventional systems, files are typically shared through the idiom of copy–modify–merge; each user obtains a copy of the current version of the file, makes their modifications, and once all changes have been made the set of modified files are merged into one single newer version. After merging, which can a be time consuming and complicated process, the file is checked back into the code repository.

Regardless of the mode of work, a single version of a file may undergo complicated concurrent changes. Using figure 4 to illustrate, a change in area A poses no likely modification problems. For regions B and C, however, it is highly likely that any concurrent changes will conflict when using a code repository system. Even if the syntax of the two changes do not directly interfere, it is likely that

the changes will cause a *merge conflict*, where the tools to produce the new version of the file are not able to reconcile the modifications of each developers.

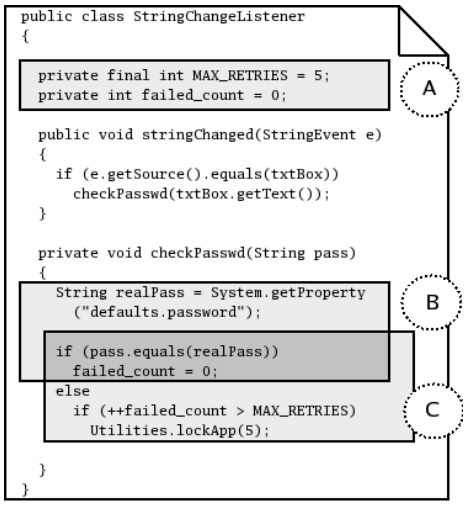


Figure 4: Artifact life-cycle within a software project.

The process of file merging is normally performed on a character–by–character basis, where no effort is made to analyse the syntax or semantics of the modified source files. Once conflicting changes have been successfully merged into a new version, the resultant file is committed into the main repository and distributed to all developers. When using CAISE to share a given version of a file, however, merge conflicts are able to be avoided altogether. We describe this further in section 3.4.

3.3.3 Transactional Conflicts

It is evident that the life-cycle of any given artifact within a project is complex. Independent of within-file concurrent modification, a further problem exists during times of between-file concurrent modifications. This issue, which we term *transactional conflicts*, occurs when a modified file, whilst syntactically valid, causes the project build to inevitably fail. An example of this is where 'User A' adds a new call to method 'I' from file 'X', while at the same time 'User B' renames method 'I' to 'J' in file 'Y'. As both modifications are syntactically valid, no syntax errors will occur within each developer's tool set. Additionally, within conventional tools neither user will detect a problem until all the new versions of the modified files are distributed to each developer and a project rebuild is attempted.

Such conflicts are normally at the considered at the scope of project level rather than at the file level. While discussing the artifact life-cycles, however, it is important to realise that transactional conflicts have an impact on artifacts—the artifacts involved require subsequent modification to enable the project to build again, even if this means reverting one or more of the recent changes. Both conventional and CAISE-based project are susceptible to transactional conflicts within a collaborative software engineering project. CAISE, however, can detect

and alert participating tools about new transactional conflicts immediately, as discussed further in section 3.4.

3.4 Typical CAISE-Tool Usage

In the previous section we described the two key types of concurrency problems that can occur during collaborative development: merge and transactional conflicts. Within CAISE-based tools merge conflicts are avoided; only one central instance of each file exists and all artifacts are shared in real time through a distributed system.

A negative aspect of file sharing via code repository systems is that merge conflicts are highly likely, even if the pending modifications appear relatively disjoint. Given the same scenario in CAISE-based tools, developers can simply observe the changes being made by other developers within the same file as they make their own modifications. These observations can be made through the components described in section 3.2 such as the real-time code editor, the artifacts panel, the user tree, the feedback panel, or any custom type of awareness support. The jitter of concurrent changes within the same region of code may appear mildly distracting when compared to working in isolation, but this is offset by the added awareness of the actions of others, and the avoidance of the costly merge processes.

In the previous section we also explained that transactional conflicts are another pitfall of CSE, regardless of the mode of work. Working with collaborative tools, however, gives the advantage of immediate discovery of transactional conflicts. As CAISE is continually modelling and evaluating the current state of the software project, the instant the project is broken feedback is provided to the developers concerned, highlighting the problem. In a situation where a code repository system is used, transactional conflicts will only be exposed once a full synchronisation of files and a system rebuild is performed, which is typically a once-a-day process for most software development organisations.

To further illustrate typical usage of the CSE tools presented in this paper, demonstrations of the tools as they execute numerous conflicting tasks are available from `www.cosc.canterbury.ac.nz/clc/cse`. These demonstrations include a scenario where the tools operate in conventional mode using a code repository system to synchronise files between users.

4 The CAISE Tool Interface

The CAISE server exists as a shared engine for participating CSE tools within a project. This section discusses how a CSE tool accesses the CAISE server, the functionality that the CAISE server provides, how a CAISE-based tool should be implemented, and how the CAISE server and associated tools can be extended.

4.1 Services Provided to CAISE Tools

CAISE-based tools are constructed rapidly in comparison to the implementation of a CSE tool where a supporting library of collaborative routines does not exist. By utilising the CAISE framework, CSE tools can rely on the CAISE server to manage the storage and sharing of artifacts, and control users as they join and

leave projects and artifacts. CAISE also provides the low-level mechanisms to allow distributed messaging between tools and the CAISE server, and supports a distributed event model.

A semantic model of the software for each project model is also maintained by the server, which is refined upon the actions of participating CAISE tools. This implies that CAISE-based tools are not required to perform any parsing or semantic analysis themselves; the server is responsible for translating modifications in artifacts to an updated semantic model. The model, however, is accessible by CAISE tools both for reading and direct modification if required.

The functions provided by the CAISE server, both in terms of supporting collaborative work and performing core software engineering tasks, allow the CSE tool developer to focus on the specific requirements of the given tool rather than re-implementing the functionality common to most CSE tools. If, however, the tool being developed requires additional features, the CAISE framework is easily extended to accommodate new artifact types and kinds of feedback. Section 4.3.2 discusses this concept further.

4.2 The CAISE Tool API

The CAISE Tool API (CTA) is provided as the means of accessing the functions of the CAISE server from within a CSE application. While the CAISE server typically resides on a separate machine, the CTA allows the calling application to view the server as if it was contained within the same process; the server functions appear no different to those of any other library. The server is accessed by a set of standard method calls, data is marshalled as method return values, and catchable events are thrown whenever interesting actions occur during the development of a CAISE project.

Table 1 presents several of the key CTA methods. This is only a subset of the complete CTA, but it provides a useful overview of the programming interface.

Method	Description
Connect to Engine	Makes a new connection to the given CAISE server
Open Project	Opens an existing CAISE project
Add Artifact	Adds a new artifact to the given project
Open Artifact	Sets an existing artifact as open for a given user
Set User Location	Moves a user's cursor location within an artifact
Update Source Code	Appends a sequence of characters to an artifact
Update Parse Tree	Appends a parse tree of an artifact
Update Model	Directly manipulates the semantic model of a project
Get Model Snapshot	Returns a copy of a project's semantic model
Fire Tool Event	Allows a tool to invoke tool-specific server plug-ins
Get Event Log	Returns the complete event log for a given project
Send Chat Message	Allows users to send text messages between tools

Table 1: Some Key Methods of the CAISE Tool API.

While the CTA makes the server appear directly accessible via conventional method calls, in reality the server is shared by an unbounded number of collaborating CAISE tools. Therefore, the CTA is completely thread safe, allowing any number of threads from any number of processes to access the server con-

currently. It was essential to implement a multi-threaded API for the CAISE server, but in doing so, it makes application programming simple—developers do not necessarily need to be concerned with calling the API from only a specific thread within their application.

Invocations of methods are treated fairly at the CAISE server. In the underlying distributed system that CAISE employs for its client interface, each incoming method call is queued and then processed in sequential order. For all other pending method calls in the queue, a low-CPU blocking mechanism is used on the client side.

4.3 The CAISE Tool Protocol

While the CTA provides the essential information for tools to implement collaborative software engineering facilities, there are contractual obligations that each CAISE tool must follow in order to keep tools correctly synchronised with each other and the CAISE server. By following the *CAISE Tool Protocol*, tools are assured of staying synchronised, and the server is able to avoid concurrency issues such as deadlocks and forced rollbacks of tool requests.

For all collaborative tools, a specialised Model-View-Controller approach is used which guarantees consistency over distributed parallel edits. Requests to edit the view are captured by tools, but the view is not immediately updated. Rather, the edit is sent to the server which in turn edits the global model, and broadcasts the change to all tools. Each tool then updates its local view, including the tool that made the edit request.

To implement a CSE tool that adheres to the CAISE tool protocol, three application-level threads are typically used: a GUI thread, a worker thread, and a CAISE event listener thread. The threading model for CAISE-based tools is presented in figure 5. Most windowing toolkit libraries provide a GUI thread, and the CAISE tool API provides a CAISE event listener thread. As the worker thread can simply be the main application thread, it is unlikely that any new threads need to be created explicitly within a CAISE tool.

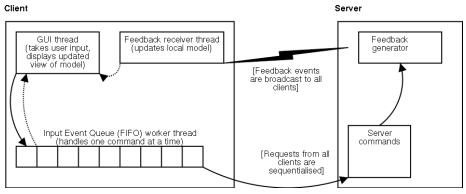


Figure 5: Threading model within a CAISE-based tool.

The following list presents the six key conditions of the CAISE tool protocol. By using a MVC approach and following the CAISE tool protocol, CSE tools are guaranteed to stay up-to-date and synchronised with the CAISE server, and there is no risk of deadlocks or loss of information.

1. The CSE tool captures all user input events such as keystrokes and caret move events, typically using action listeners. All actions are to be consumed, blocking the underlying view of the artifact such as a source file or class diagram from being modified.

2. All captured events are placed into a FIFO event queue within the CSE tool. The GUI thread returns immediately after placing the event in the queue, preventing any latency within the user interface.

3. A separate CSE tool worker thread dequeues events and issues them to the server as corresponding CTA method invocations.

4. The CSE tool worker thread waits for the return value of the CTA method invocation before processing the next tool input event. The CSE tool does nothing upon a successful method invocation, and escalates any errors if the method invocation fails.

5. The CSE tool's CAISE event listener thread listens for broadcasted server events that result from CTA method invocations. Upon relevant events such as artifact modifications and user location changes, the model of the artifact within the tool is updated accordingly. This step is performed by all participating tools, not just the instance that invoked the event.

6. Upon any model update, the CSE tool's artifact view is redrawn by the GUI thread.

During spikes of development by multiple CAISE tools, the server ensures fairness by queuing events evenly based on the inbound tool connection, rather than order of arrival. This way, we avoid the situation where all other tools are unfairly delayed by an exceptionally active single user.

4.3.1 Tool Synchronisation

Individual CSE tools have the ability to implement locks and other floor control policies that allow only one user at a time to edit a given region of code. By default, however, the CAISE framework allows full synchronous editing of any artifact. To ensure that tools are always synchronised, the CAISE tool protocol need to handle the case where seemingly conflicting edits between tools are accommodated. Accordingly, relative user positions are supplied in all communication between the tools and the server, which means that interleaved edit events will produce the same buffers in all views.

Absolute user positions are also used within the framework, but these are only determined after an input event has been processed by the server and broadcast back to all participating tools; prior to event serialisation by the server, tools have no way of asserting that there are no other pending edits to the same artifact, which has the potential to skew the location of the edit request. Absolute user positions are normally used for synchronisation purposes only; to ensure that the CAISE tool's buffer is synchronised with the server, the tool checks the current absolute position of the user within the given local view of the artifact, and verifies that this matches with the last reported absolute position as indicated by the server.

4.3.2 Tool Manager Modules

The CAISE framework provides generic support for the collaborative editing of text documents, the parsing of source files, and the semantic analysis of parse trees derived from source code and UML diagrams. Often, however, tools require

further functionality from the server, including the support of new artifact types. To accommodate extensibility within the CAISE server, modules known as *Tool Managers* can be integrated through the CAISE plug-ins interface.

If we take the UML class diagrammer presented in figure 2, it is apparent that such a tool contains more information than what is provided by the semantic model such as a list of all classes and methods. A class diagrammer contains class layout information that must be shared every time any instance of the class diagram is modified. Therefore, when implementing this UML diagramming tool, an additional type of artifact was able to be stored, modified and shared by all participating tools via a tool manager module specifically written for the UML diagrammer.

The CAISE server does not have any knowledge about the structure or semantics of new types of artifact that tool managers introduce. Rather, the tool manager relies the CAISE server only to store the artifact, and the manger itself responds to modification requests as invoked by the UML diagramming tool.

In the case of the UML diagramming tool, whenever a user requests, for example, to change the location of a displayed class, a tool-specific event is thrown to the CAISE server via the CTA, and this event is simply proxied to the UML diagrammer tool manager. The tool manager will then access and update the artifact that stores the class location information, and then broadcast this change out to all tools in exactly the same manner as the core CAISE tool protocol, allowing all users to update their shared view of the UML class diagram.

The CAISE server can be extended in many other ways through the plug-ins interface, including support for new languages. For further details please refer to [9].

5 Activity Management

This section discusses how CAISE and its supporting tools address aspects of large-scale software engineering related to the management of large code bases and teams of users.

5.1 Working from a Source Code Repository

From a viewpoint of real-time CSE, source code repository systems may at first appear antithetical to CSE tools; the purpose of tools support for CSE is to enable developers to work together, not to partition themselves. While this argument is certainly true for small development groups, within the realm of open-source software development the use of source code repository systems is essential and unavoidable.

Users of the CAISE architecture are still able to work collaboratively within a code repository controlled environment. This is achieved by forming a group of collaborating users, and working collaboratively through the CAISE architecture within this group. The set of source files within the CAISE project will be based from the latest version from the code repository, and this collaborative group will be required to periodically re-synchronise their code-base with that of the central repository.

This approach still subjects the collaborative group to the same problems that the code repository users face: merge and transactional conflicts upon re-synchronisation. However, the collaborative group benefits from having the ability to work together within the group. Additionally, if the area that the collaborative group is working on within the project is loosely coupled from the rest of the project, merge conflicts should largely be avoided, and transactional conflicts are also likely to be low.

The success of this approach depends on the number of collaborative groups within the entire project, the size of each collaborative group, the ratio of collaborative to conventional developers, the degree of coupling between packages within with project, and the development approach of the programmers.

5.2 Partitioning of Large Projects

The CAISE infrastructure is best suited to a single group of developers working collaboratively on an entire project. For the development of large projects, however, often developers will work within separate areas of code, especially when the number of developers is large and several tasks can be performed concurrently.

In a well designed large software project, there are likely to be separate areas for developers to focus on, and a natural partitioning of roles can take place. A simplistic example of such a project is presented in figure 6. Within professional development groups, well partitioned projects and structured development approaches are likely. In this situation, where very few changes within a partition should affect the development efforts of those working on other areas of code, development *crosstalk* is likely to be at an acceptably low level. In this case, the use of real-time CSE tools is also suitable.

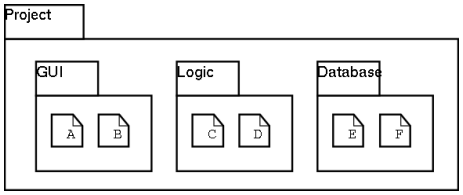


Figure 6: A simplistic example of a well partitioned software project.

In some large projects, however, even if they are well partitioned, it is possible that the developers will prefer no crosstalk from the programmer activity within other partitions of the project. In this situation, it is still possible to accommodate collaboration within each development partition by using the code repository mechanism discussed in section 5.1. In the latest version of CAISE, tool support exists for the partitioning of projects in this manner via the code repository interface.

By partitioning a group of developers within a project into subgroups, crosstalk between groups is eliminated. The trade-off, of course, is that communication between groups is reduced and a synchronisation process must take place at regular intervals between groups. However, if the project is well designed and

the developers use a structured software engineering approach, merge and transactional conflicts are likely to be low.

This hybrid approach, as described in the middle segment of figure 7, implements project partitioning, where crosstalk is likely to be less than in full collaborative mode, and merge conflicts and transactional conflicts are likely to be less than in the conventional, code repository mode. We stress, however, that this approach should only be used when groups of developers purposely wish to separate themselves.

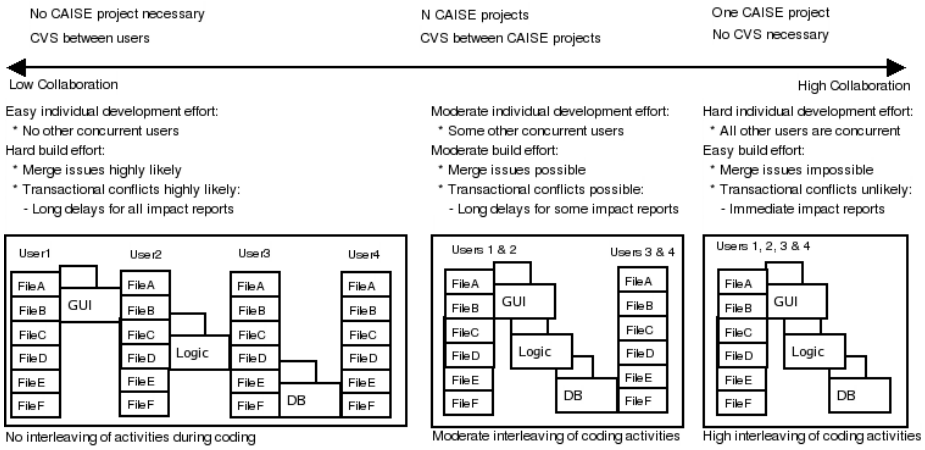


Figure 7: The CAISE collaborative usage spectrum.

At the conventional development end of the spectrum, a code repository system allows each user to have his or her own code-base to work on, and usually each programmer will also try to modify only the subset of files within his or her current area of focus. In this configuration, the individual programming effort is relatively easy, but transactional and merge conflicts are likely.

At the other end of the spectrum, using a single CAISE project negates all use of code repositories. This implies that while higher levels of development crosstalk are possible, transactional conflicts are less likely and merge conflicts are completely avoided. We argue that for most small and medium sized development groups it is better to work as one collaborating team—development jitter during spikes of activity is preferable to ongoing conflicts and reduced programmer communication between developers.

5.3 Compilation Crosstalk Avoidance

Unexpected real time code modifications by other users, whilst surprising, do not significantly degrade a developers ability to work within a collaborative setting. If one developer is working on the same line of code as another developer, it is likely to be beneficial if both parties pause and discuss the current activities, although programmers may choose to ignore the presence of others and carry on development. A major problem with real time development, however, is that of compiling code during a time of concurrent development activity.

Ideally, if one developer makes a change that is unrelated to the area of the program that another developer is currently working on, the second developer should not necessarily be placed in a position where he or she is prevented from compiling. This principle, known as private work, is one of the key elements to CVS and related repositories. For CSE tools, however, if the first developer has not completed their changes, or their changes are syntactically or semantically incorrect, the project will fail to complete its build even for the second user as the entire project is shared in real time. To resolve this problem, we have refined the project build panel with a special *collaborative view* feature, which is presented in figure 8.

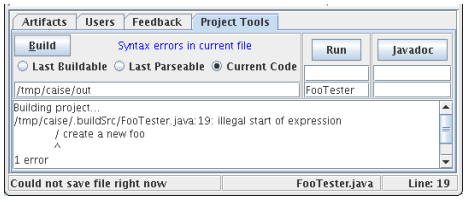


Figure 8: Tools Panel with adjustable levels of project crosstalk.

The view facility within the project tools panel allows compilation to take place from within three different modes: *current*, *last parseable* and *last buildable*. These modes are depicted in figure 9. In current mode, the panel attempts to build the latest version of the code, which will fail if any recent remote changes have broken the build. In last parseable mode, the build only takes into account the last syntactically correct version of each file. This way, if a remote programmer is current editing a file, the changes will only take affect once the code is properly formed. In last buildable mode, the panel will produce an executable based on the last version of the program that had no build errors.

This has proved to be a particularly useful feature for real collaborative software engineering, and we suggest it should be employed as a global strategy. It may provide an alternative to partitioning a collaborative group in the case of exceeding a tolerance threshold for remote user activity.

5.4 Private Work

The concept of CAISE is about enabling developers to work together. Private work, where developers modify a code base independent of other concurrent developer activities, may appear antithetical to the principles of CAISE, but it is on some occasions essential in real-world software engineering scenarios.

If a given developer deems it essential to work in isolation for a considerable amount of time, he or she can work on an alternative branch of the code-base using a code repository system, and merge the changes back into the main CAISE project upon completion. Following the conventions of code repository based software engineering, it is good practice to discourage all other users from modifying their version of this file if possible during this period.

CAISE-based CSE tools have the potential to implement a private work mode, where the changes of others are prevented from being propagated to the isolated developer's tool-set. This feature would be implemented in a man-

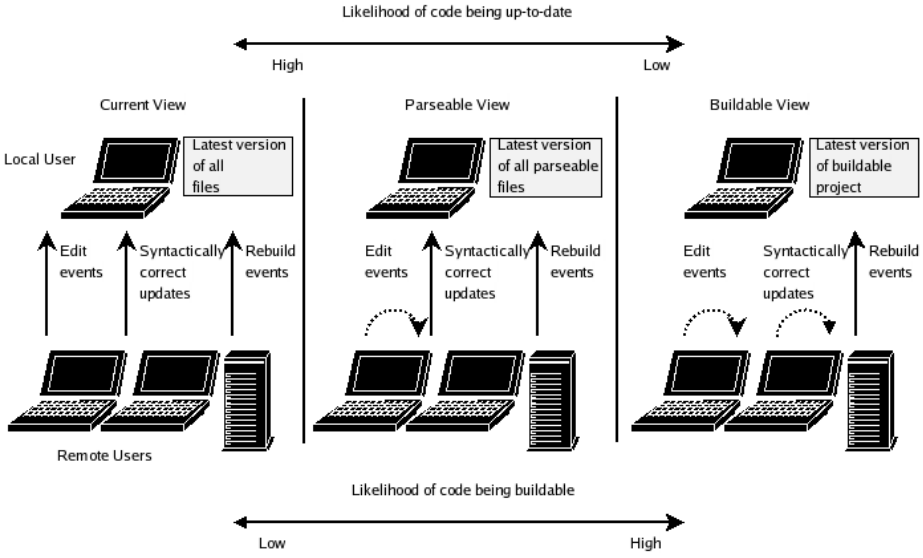


Figure 9: The various modes of collaborative view when compiling from within a CAISE tool.

ner that is similar to the compilation crosstalk avoidance mechanism. Unfortunately, such a tool mode has the potential to attract merge conflicts upon reintegration with the main CAISE project, and draws away from the whole concept of collaborative work. Therefore, the private work mode within our set of CSE tools has not been instantiated. Designers of other CAISE tools are free however to implement such features.

6 Performance Analysis

A final consideration when discussing the design and use of collaborative tools for software engineering is that of performance. The performance of the tools must be satisfactory, and there should be no theoretical limitations of the architecture that will prevent the tools from being useful in realistic environments. While the core response speeds and resource usage of CAISE and its supporting tools have proved acceptable over a long period of subjective testing and user evaluations, it is important to note the effects of code base size and number of concurrent developers on server memory load and tool response times.

6.1 Memory Load

To provide features such as code modification impact reports and degree of interest feedback, the CAISE server maintains a semantic model of the software within the project. An immediate concern is that of memory usage; if a large amount of memory is required for each line of code added to the model, projects of a realistically large size might be beyond the scope of the CAISE architecture.

Figure 10 presents the amount of memory used per line of code across a

range of CAISE projects. For any CAISE based project, the server first loads in all packages, classes, interfaces and methods directly accessible from any Java source file. This brings the initial project model size up to around 60 Mb. From that point onwards, however, most of the components that the modelled software rely upon are now loaded, and the project model size increases only linearly in relation to the number of classes and methods declared in each source file. After taking the project initialisation into account, each line of code adds approximately a kilobyte of server memory.

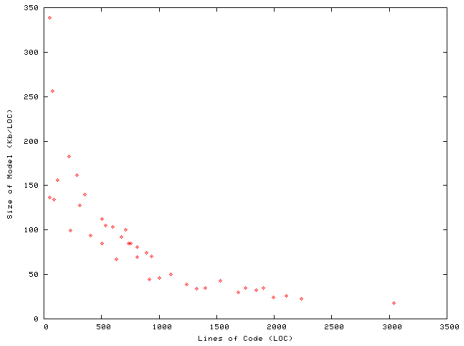


Figure 10: Lines of Code versus Server Memory Usage.

For large software projects where there can be potentially millions of lines of code within a single revision, an alternative to an in-memory model might be required. In commercial settings, it is likely that specialised hardware can support multiple gigabytes of memory. In other situations where mass memory capabilities are not available, the CAISE architecture can easily be extended to incorporate an object-oriented database for models of potentially any size.

While the memory requirements for a CAISE-based project may seem significant, it is important to note that no other demands are placed on memory resources throughout the entire development environment. Unlike other architectures including IDEs, each CSE tool can rely on the CAISE server for all parsing, analysing and modelling of the software; tools themselves do not need to store a replica model.

6.2 Network Load

The design of the architecture ensures that network loads are as low as possible, and recent analysis of traffic verifies that for small user groups, no considerable strain is placed on a 100 Mbps Ethernet local area network. Even as the number of concurrent users increases to that of large development teams, today's networks are capable of accommodating the load.

When testing on wide area networks, the data throughput requirements are low enough for clients to be connected to the server from dial-up networks, but the latency can cause edit delays of up to several seconds. To support low speed wide area network connections, an alternative distributed system might be necessary where the anticipated results of modification requests are immediately displayed in the originating tool's display. In this case, a synchronisation

routine will be required to run in a separate thread to resolve any modification discrepancies between tools.

6.3 Response Times versus Number of Users and Model Size

We are confident that as the number of users and the size of the project model increases, response times will remain stable. The direct impact of increased numbers of concurrent users within a CAISE project is negligible; the number of connected users or opened files does not have a noticeable effect on server memory usage or response time. If all users are highly active at the same time the server response times will slow down during this period, but in reality this is a very unlikely scenario.

Even if a given project has a very large semantic model, this does not necessarily affect the response times of the server. Most operations such as adding a new method to a class or querying the model for a specific relationship only require the traversal of a fixed subset of the entire model space. Therefore, even as the model grows in size, the response times should stay approximately constant.

7 Conclusions and Future Work

Real-time support for CSE is an important emerging field of research. The size and complexity of today's software projects is far exceeding the ability of conventional single-user tools to provide an environment of fine-grained communication between developers. While source code repository systems provide a degree of control over constantly evolving software projects, there is both the demand and enabling technology for more comprehensive tool support.

We have developed a framework, CAISE, to support CSE. In this paper, we have illustrated how CAISE-based CSE tools can be integrated into real-world development scenarios. This is a significant step towards advancing CSE tools from research prototypes towards essential elements of software engineering practice, allowing the perceived benefits and opportunities of CSE to be realised.

To date, the large development cost in constructing new CSE tools has been a major obstacle within the field of research. In this paper, we have shown how new CSE tools can be developed rapidly within the CAISE framework. We have also shown how CSE tools can be used in various development scenarios, to suit the nature of the project and the development methodologies of the team.

Finally, in this paper we have addressed various performance issues and have illustrated that CAISE-based CSE tools are able to operate satisfactorily under a range of workloads.

The CAISE framework and associated tools have also performed well during recent empirical and anecdotal evaluations. After illustrating the construction and use of CAISE-based CSE tools in this paper, it is hoped that others are encouraged to develop similar tools to support the emerging field of real-time CSE.

References

[1] Brian Berliner. CVS II: Parallelizing Software Development. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 341–352, Berkeley, CA, 1990. USENIX Association.

[2] Marko Boger, Thorsten Sturm, Erich Schildhauer, and Elizabeth Graham. *Poseidon for UML User Guide*, 2002. URL `http://www.gentleware.com/support/documentation.php4`.

[3] R. P. Carasik and C. E. Grantham. A Case Study of CSCW in a Dispersed Organization. In *CHI '88: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 61–66, New York, NY, USA, 1988. ACM Press. ISBN 0-201-14237-6.

[4] Carl Cook. Collaborative Software Engineering: An Annotated Bibliography. Technical Report TR-COSC 02/04, Department of Computer Science and Software Engineering, University of Canterbury, Christchurch, New Zealand, June 2004. Work in Progress.

[5] Carl Cook. *Towards Computer-Supported Collaborative Software Engineering*. PhD thesis, University of Canterbury, Christchurch, New Zealand, December 2005. Work in Progress.

[6] Carl Cook and Neville Churcher. An Extensible Framework for Collaborative Software Engineering. In Deeber Azada, editor, *Proceedings of the Tenth Asia-Pacific Software Engineering Conference*, pages 290–299, Chiang Mai, Thailand, December 2003. IEEE Computer Society.

[7] Carl Cook and Neville Churcher. A User Evaluation of Synchronous Collaborative Software Engineering Tools. Technical Report TR-COSC 04/05, Department of Computer Science and Software Engineering, University of Canterbury, Christchurch, New Zealand, August 2005.

[8] Carl Cook and Neville Churcher. Modelling and Measuring Collaborative Software Engineering. In Vladimir Estivill-Castro, editor, *Proceedings of ACSC2005: Twenty-Eighth Australasian Computer Science Conference*, volume 38 of *Conferences in Research and Practice in Information Technology*, pages 267–277, Newcastle, Australia, January 2005. ACS. 25% acceptance rate.

[9] Carl Cook, Neville Churcher, and Warwick Irwin. Towards Synchronous Collaborative Software Engineering. In *Proceedings of the Eleventh Asia-Pacific Software Engineering Conference*, pages 230–239, Busan, Korea, December 2004. IEEE Computer Society.

[10] Jon Froehlich and Paul Dourish. Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams. In *6th International Conference on Software Engineering (ICSE'04)*, pages 387–396, Edinburgh, Scotland, United Kingdom, May 2004. IEEE.

[11] Nicholas Graham, Hugh Stewart, Authur Ryman, Reza Kopaee, and Rittu Rasouli. A World-Wide-Web Architecture for Collaborative Software Design. In *Software Technology and Engineering Practice*, pages 22–32, Pittsburgh, Pennsylvania, August 1999. IEEE.

[12] Michael Reeves and Jihan Zhu. Moomba A Collaborative Environment for Supporting Distributed Extreme Programming in Global Software Development. In Jutta Eckstein and Hubert Baumeister, editors, *Lecture Notes in Computer Science*, volume 3092, pages 38–50. Springer-Verlag, January 2004.

[13] Anita Sarma and Andr van der Hoek. Palantr: Coordinating Distributed Workspaces. In *26th Annual International Computer Software and Applications Conference*, Oxford, England, August 2002. IEEE.